

Being Cloud-Native, Not Naïve

CNAE – Cloud Native Architecture & Engineering



Prof. Stefan Tai
Sebastian Werner, Elias Grünewald, Nicola Leschke, Maria Borges

Information Systems Engineering
TU Berlin

Recap: Software System Qualities (aka Non-functional Requirements)

Nonfunctional Requirements		
Accessibility	Efficiency	Reliability
Availability	Extensibility	Responsiveness
Compatibility	Failure Transparency	Robustness
Certification	Fault Tolerance	Safety
Compliance	Flexibility	Scalability
Configurability	Interoperability	Securability
Dependability	Maintainability	Supportability
Deployability	Operability	Testability
Documentation	Performance	Traceability
Disaster Recovery	Recoverability	Usability

➔ How can we achieve these qualities? Each single one, multiples, all?

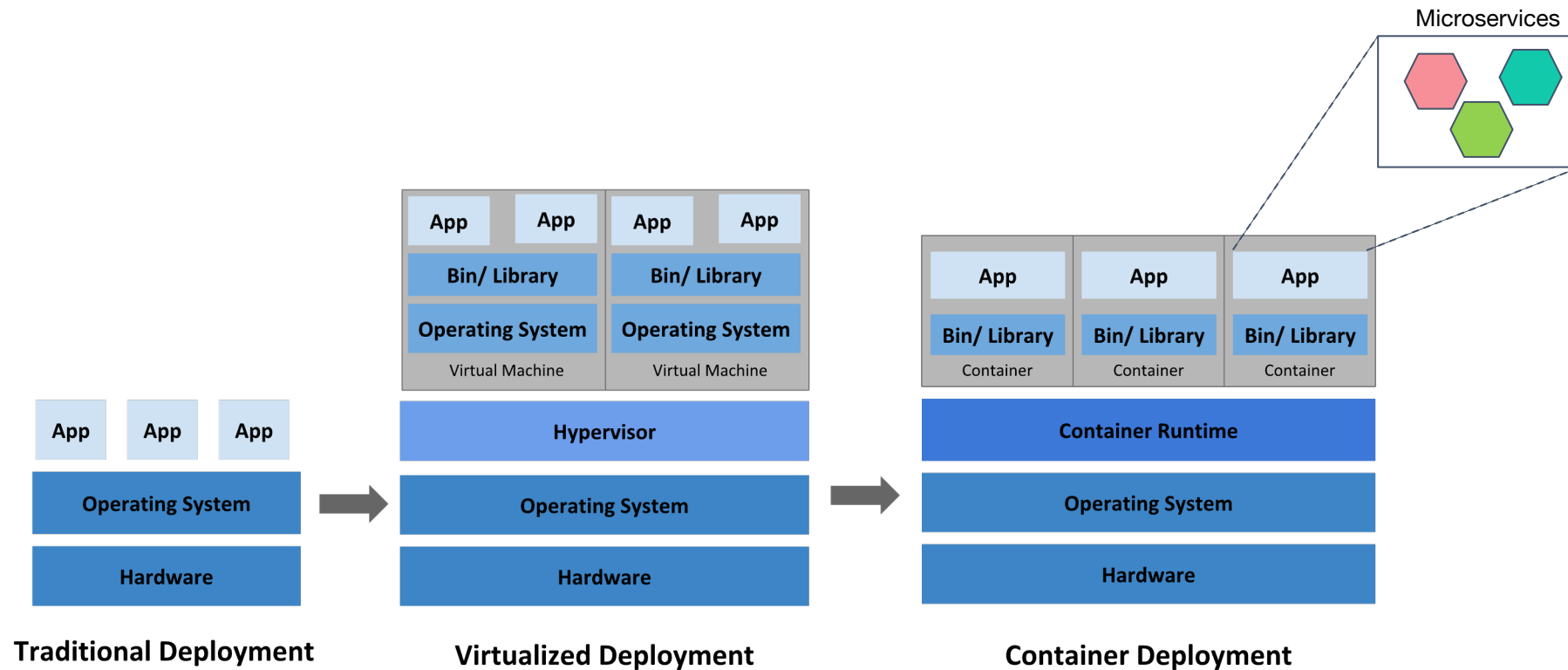
Recap: Microservice Architecture and Principles

“In short, the microservice architectural style is an approach to developing a single application as a suite of **small [focused] services**, each **running in its own process and communicating with lightweight mechanisms**, often an HTTP resource API.

These services are **built around business capabilities** and are **independently deployable** by **fully automated deployment machinery**.

There is a **bare minimum of centralized management** of these services, which may be written in **different programming languages** and use **different data storage technologies**.” [Martin Fowler]

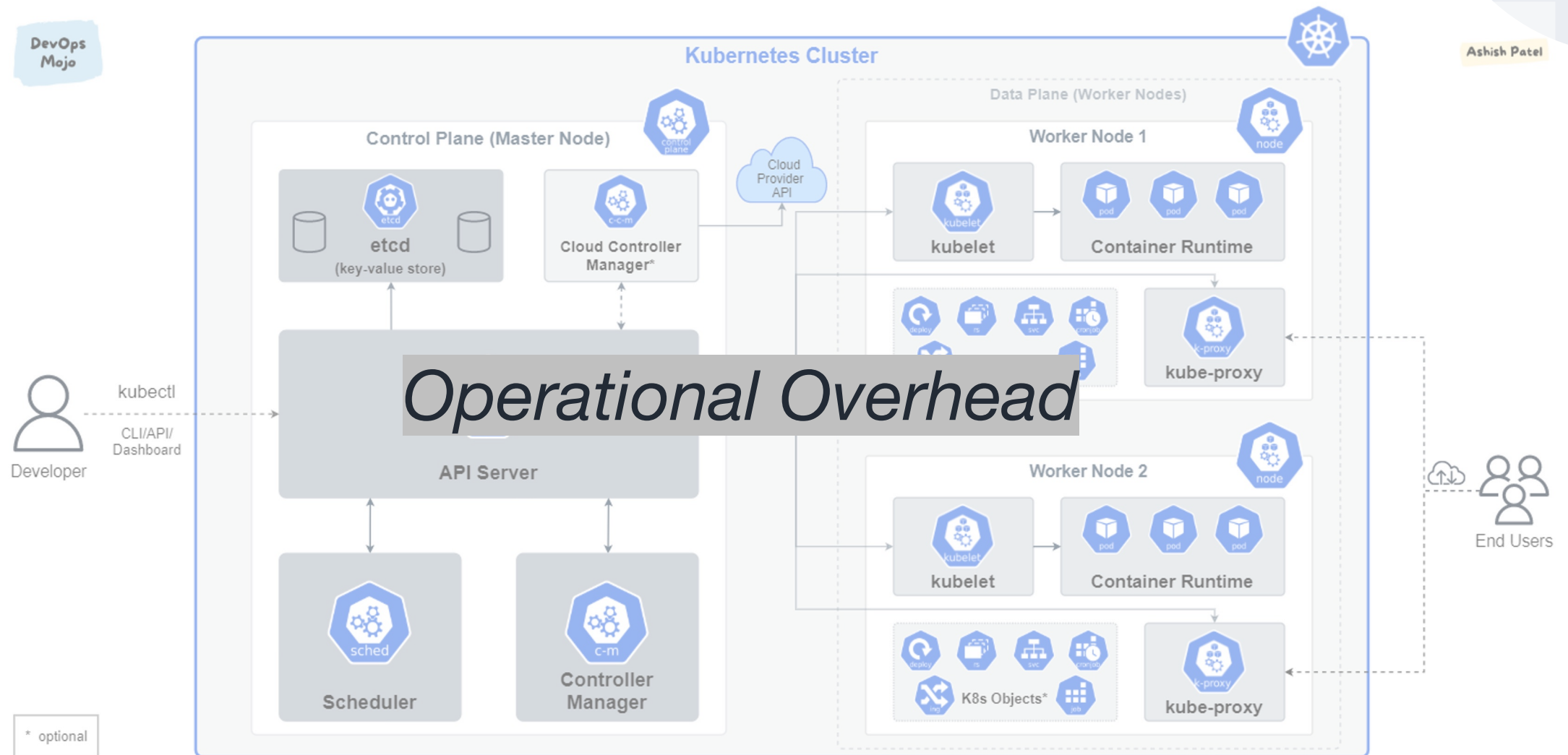
Recap: The Role of Containers (and Container Orchestration)



Source: <https://kubernetes.io/docs/concepts/overview/>

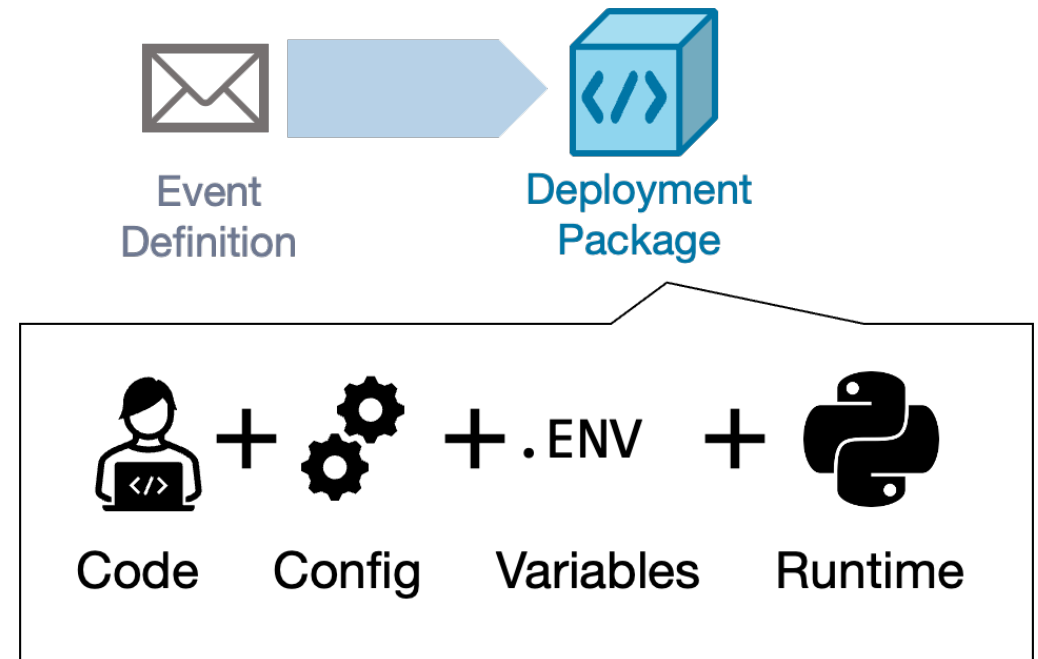
- With containers, provisioning takes seconds instead of minutes, with less configuration chaos
- However, deployment at-large requires orchestration of multiple containers, introducing new challenges

Recap: Challenges of Container-backed Deployments



Recap: Serverless Applications (aka “Not worrying about servers”)

- Instead of managing resources, a user provides a task description and the cloud automatically provisions resources to perform that task.
- “Serverless models abstract away most of the DevOps related concerns.”
- A serverless service can be controlled with a limited set of configurations that govern application quality and cost.
- Typically, glue code/infrastructure between services and applications.
- Very easy to use, start with – hard to get right, hard to maintain.

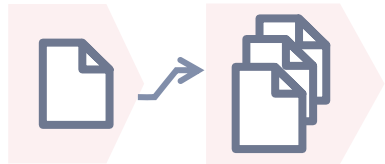


Recap: Serverless Assumptions vs. Realities



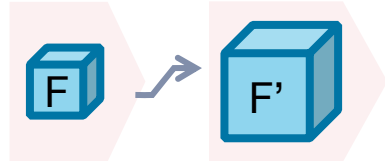
Function abstraction matches all workload.

Workload must match function handler configurations.



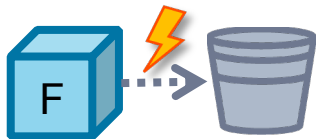
Functions scale automatically with incoming events.

Provisioning time may not suit scaling app needs.



Fully managed ops tasks solve diverse ops needs.

Fixed strategies for applying ops tasks.



Functions can easily be composed to implement any business logic.

Function compositions experience complex limitations and add costs.



FaaS offers significant advantages for execution cost.

Marginal execution costs are higher, and compositions are cost drivers.

Being Cloud-Native, not Cloud-Naive

"Cloud-Native" is about designing, developing and running applications so that they take full advantage of the cloud.

Designing... modular, API-driven systems, with lightweight, stateless interactions

Developing... loosely-coupled systems, often microservices-based, ie. with independent scalability and decentralized data management

Running... observing (and learning from) actual operations including faults, reacting to actual varying (and possibly predicting) workloads, optimizing costs

Taking advantage... of diverse, alternative cloud features and service offerings, to address and ensure target system qualities

Looking Back at the Initial Question of Quality

Nonfunctional Requirements		
Accessibility	Efficiency	Reliability
Availability	Extensibility	Responsiveness
Compatibility	Failure Transparency	Robustness
Certification	Fault Tolerance	Safety
Compliance	Flexibility	Scalability
Configurability	Interoperability	Securability
Dependability	Maintainability	Supportability
Deployability	Operability	Testability
Documentation	Performance	Traceability
Disaster Recovery	Recoverability	Usability

➔ How can we achieve these qualities? Each single one, multiples, all?

(Cloud-native) Architecture

“[A]rchitecture is selling options”

e.g. “What size of server do you need to purchase for a system? If your application is architected to be horizontally scalable, this decision can be deferred: additional servers can be purchased later, at a known unit cost.”

Gregor Hohpe

<https://architectelevators.com/architecture/architecture-options/>



Watch out for our upcoming guest lecture by the AWS User Group!

Weighing Options

“An architect's job therefore is to consider architecture options in a broader context, which can involve diverse factors such as commercial and legal agreements, skills availability, or an installed base.”

Gregor Hohpe

<https://architectelevators.com/>

<https://martinfowler.com/articles/architect-elevator.html>



Watch out for our upcoming guest lectures by DATEV and HUK!

By focusing on select qualities, new sub-disciplines emerge,
e.g., Site Reliability Engineering (SRE)

“SRE is what you get when you treat operations as if it’s a software problem. Our mission is to protect, provide for, and progress the software and systems [...] with an ever-watchful eye on their availability, latency, performance, and capacity.”

<https://sre.google>



Watch out for our upcoming
guest lectures by Google!

When Quality Priorities Change: Architecture Refactoring

	Code Refactoring	API Refactoring	Architectural Refactoring
Improves	Internal structure, maintainability	Interface, many quality attributes	Overall structure, many quality attributes
Target	Source code	Specification, source code, API implementation components	Components, (sub)-systems, documentation, architectural decisions
Impact	Internal only	Externally visible, API clients	Potentially large group of stakeholders (3 rd parties, other teams, operations), often system-wide or cross-cutting
Drivers	Code smells, test driven development	Changed client- and provider-side quality requirements, API smells	Changed requirements and constraints, architectural smells
Evolution	Not key, observable behavior unchanged	Important, compatibility with existing clients	Depends on visibility of changes
Examples	Rename, Extract Method, Move Class	Split Endpoint, Inline/Extract Information Holder	Migrate from SQL to NoSQL («De-SQL»), Split Subsystem

Table Source: M. Stocker & O. Zimmermann. From Code Refactoring to API Refactoring: Agile Service Design and Evolution. Springer CCIS, Vol. 1429, 2021.



Watch out for refactoring insights gained by all student teams!

Looking Back at the Initial Question of Quality

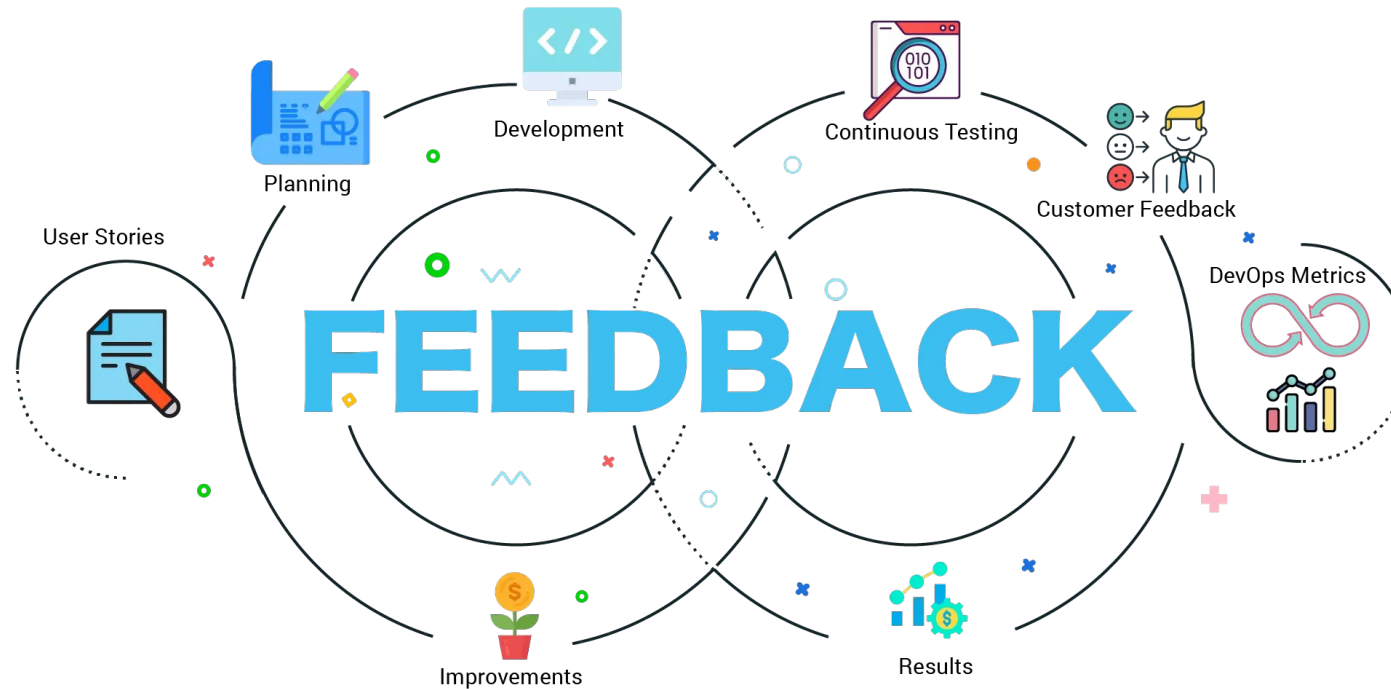
Nonfunctional Requirements		
Accessibility	Efficiency	Reliability
Availability	Extensibility	Responsiveness
Compatibility	Failure Transparency	Robustness

“You can’t control what you can’t measure”

Configurability	Interoperability	Securability
Dependability	Maintainability	Supportability
Deployability	Operability	Testability
Documentation	Performance	Traceability
Disaster Recovery	Recoverability	Usability

➔ How can we achieve these qualities? Each single one, multiples, all?

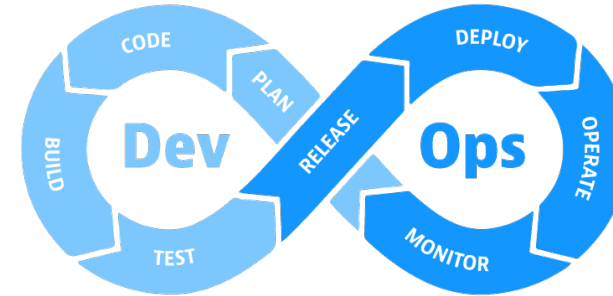
Validate Decisions through Feedback Loops



What methods and tools can we use to measure “quality”?

Ensuring Runtime Quality – Example: Monitoring

Recall DevOps: *Monitor!*



Monitor:

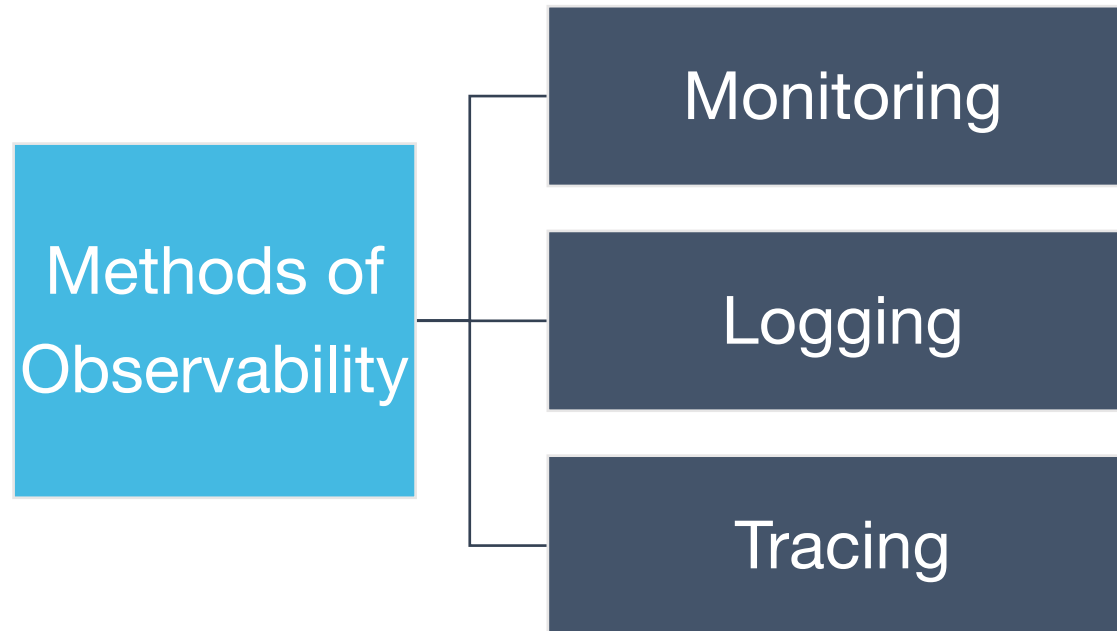
“A software tool or hardware device that operates concurrently with a system or component and supervises, records, analyzes, or verifies the operation of the system or component.”

→ (1990) IEEE Standard Glossary of Software Engineering Terminology

Monitoring tools in SE are specialized software, observing a component/system/service at runtime

A monitoring tool everyone knows: OS task manager

Is it just “Monitoring”?



Tracing, monitoring and logging all have the same purpose: making applications **observable**

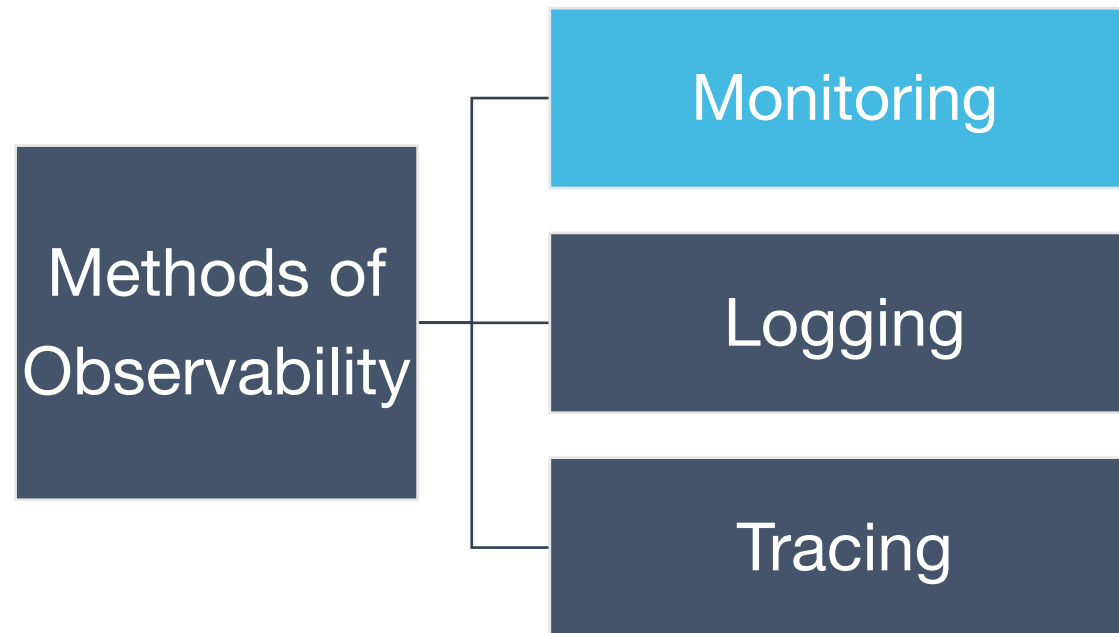
- **Observability**, as a quality, characterizes to which degree we can observe software **at runtime**
- Tools and methods of observability make software observable
 - Monitoring: „monere“ (Latin: to admonish, warn)
 - Tracing: “following a trail/path”
 - Logging: recording events in journal-like form (c.f., “logbook” from seafaring)
 - (Telemetry: „têle“ (GR: from a distance) + „meter“ (sth. that is concerned with measuring))
- Different, often imprecise terminology for the same or similar tools

Observability has trade-offs

- Higher complexity and heterogeneity make observability more costly
- Better observability can help improving quality



Observability Methods



Resource / System Monitors

Account for available and used resources at OS-level

- CPU, RAM, network bandwidth, disk capacity, ...
- Different granularity
 - Measurement interval: 1m, 1s, 1ms, ...
 - Moving average, last/current value, ...

Virtualization

- Resource monitors for physical and virtual environments
- It can be challenging to isolate observations in shared environments

Many tools exist to monitor arbitrary sizes of clusters and types of OS

Application Monitors

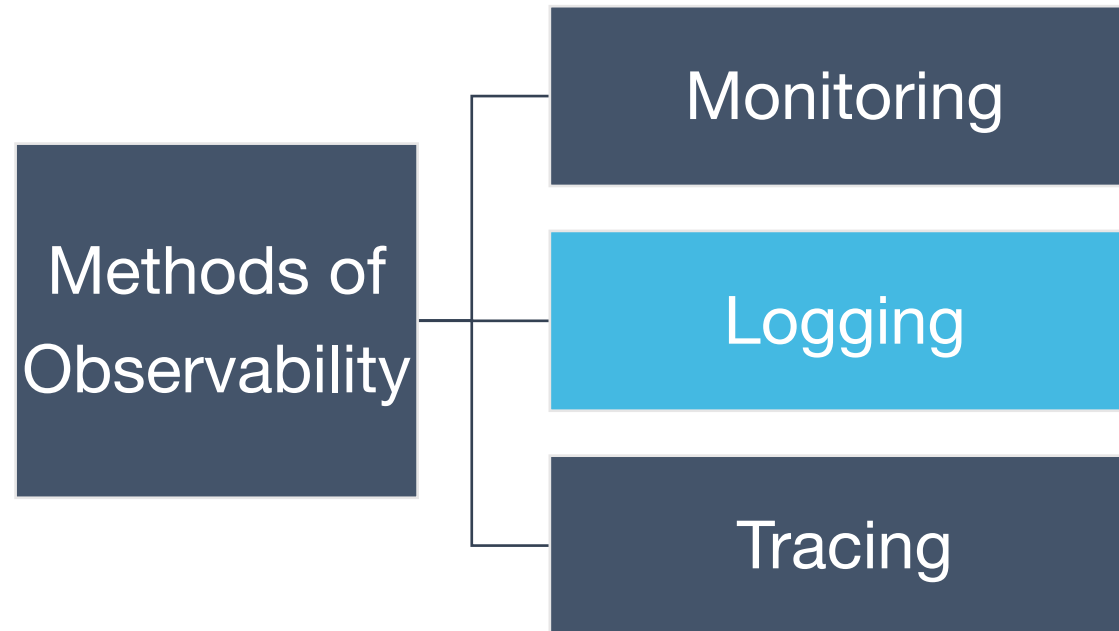
Relies on applications to actively expose metrics

- E.g., Java-based applications can expose heap memory statistics
- Exposing metrics is done through standards
 - Language- or runtime-specific, e.g. JMX
 - Framework-specific endpoints/client libraries, e.g. [Prometheus](#)

Metrics are application(-type) specific

- Web-Applications, e.g.:
 - Request-response latency
 - Response size
- Database transactions, e.g.:
 - Runtime
 - Changed rows
- Ticket booking system: reservations/minute

Observability Methods



Fundamental principle: record an event, consisting of (at least) *timestamp* and *description*

- Adding log statements is a basic activity of software development
- Logging frameworks differentiate *levels*, e.g. in Java (ERROR, WARN, INFO, DEBUG, ..)
- Logging output is often simply printed to *stdout*

Problems of “basic logging”

- Another developer → another format and Log-level?
- Textual descriptions are ambiguous
Example: “Webserver started” - is configuration completed? Is the database connection working? What port is the process listening on?
- Context is optional
 - Which class method produced the log statement?
 - Software version?
- Logs in different programming languages?
- Keeping logs and making them searchable?

Structured vs. Unstructured Logs

- *Unstructured* (plain text)

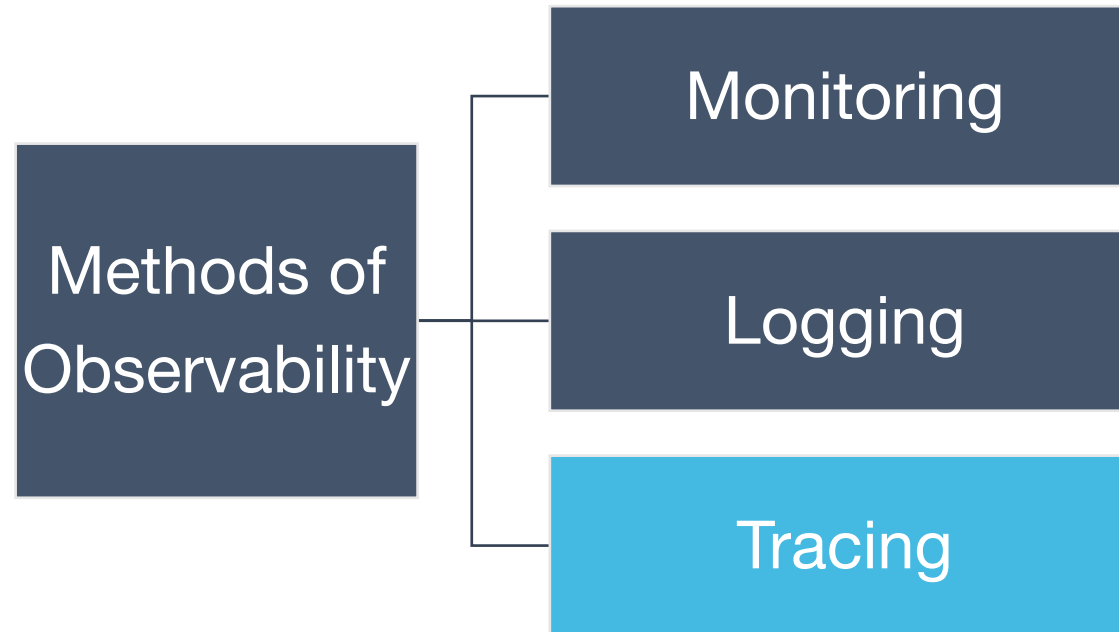
```
DEBUG 2018-11-10 16:17:58 - Incoming request from clientID 54732
```

- *Structured* (e.g. JSON)

```
{  
  "msg" : "Incoming request",  
  "data" : {"clientID":54732},  
  "timestamp" : 1541866678,  
  "level" : "DEBUG"  
}
```

Modern cloud-native architectures typically aggregate logs of different system components in a unified logging layer, which facilitates troubleshooting

Observability Methods



Distributed Tracing

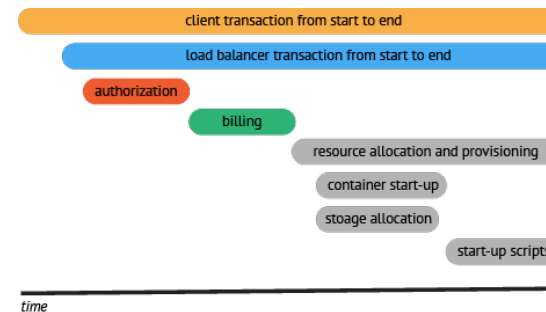
The goal of distributed tracing is to observe the **end-to-end** behaviour of distributed systems.

Follow interactions between components by correlating events.

Purpose: latency optimization, error analysis, debugging

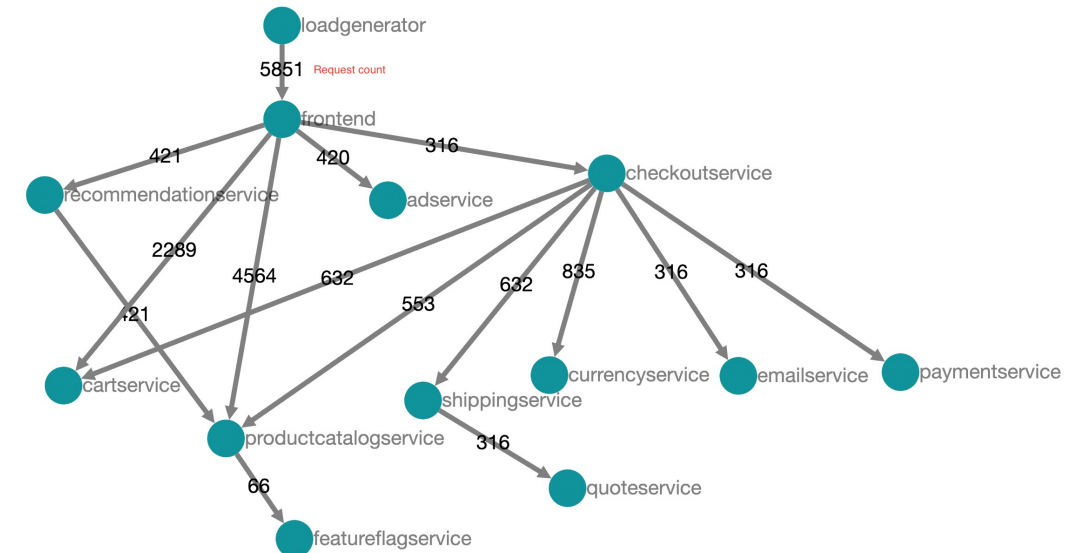
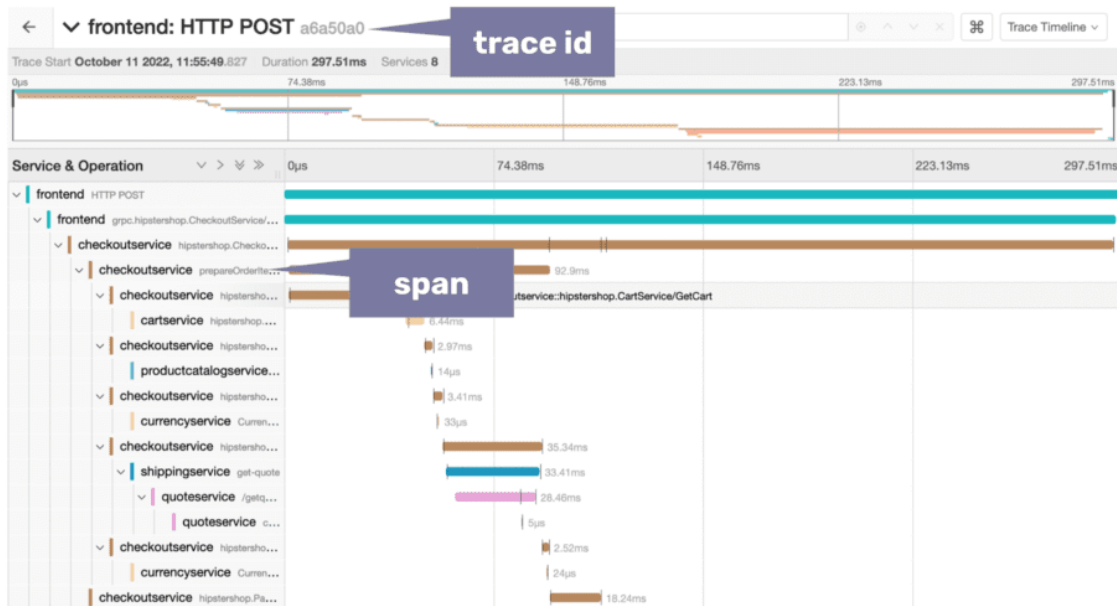
A trace consists of a graph (tree or DAG) of “spans”

- Span = logical container for **events** happening between request-response-pairs
- In composition, a trace represents a complete interaction from client perspective



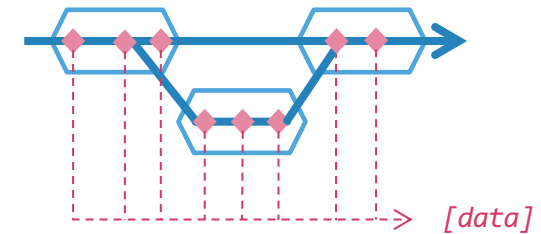
```
{
  "traceID": "c81452d4b90bb1f",
  "spans": [{
    "traceID": "c81452d4b90bb1f",
    "spanID": "c81452d4b90bb1f",
    "operationName": "root",
    "references": [],
    "startTime": 1576489836655758,
    "duration": 61587,
  },
  {
    "traceID": "c81452d4b90bb1f",
    "spanID": "f3c6aa8fcfd5f8",
    "operationName": "leaderboards",
    "references": [{
      "refType": "CHILD_OF",
      "traceID": "c81452d4b90bb1f",
      "spanID": "c81452d4b90bb1f"
    }],
    "startTime": 1576489836655783,
    "duration": 41328
  },
  ...
}]
```

Distributed Tracing – Single Trace vs. Service Graph

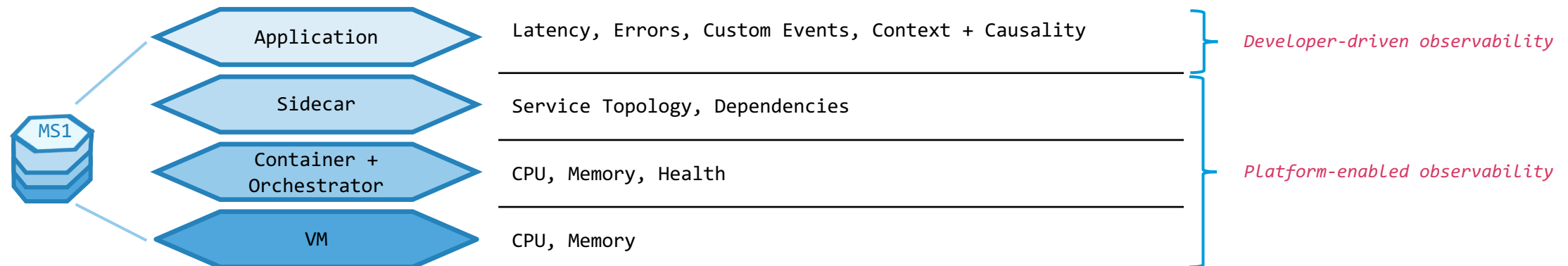


Observability requires Instrumentation

Tracing, monitoring and logging require instrumentation, i.e. points in code that produce events + data when executed



Can be achieved by developers manually instrumenting the application code or by leveraging “out-of-the-box” instrumentation of underlying platforms



K8s Observability - Monitoring

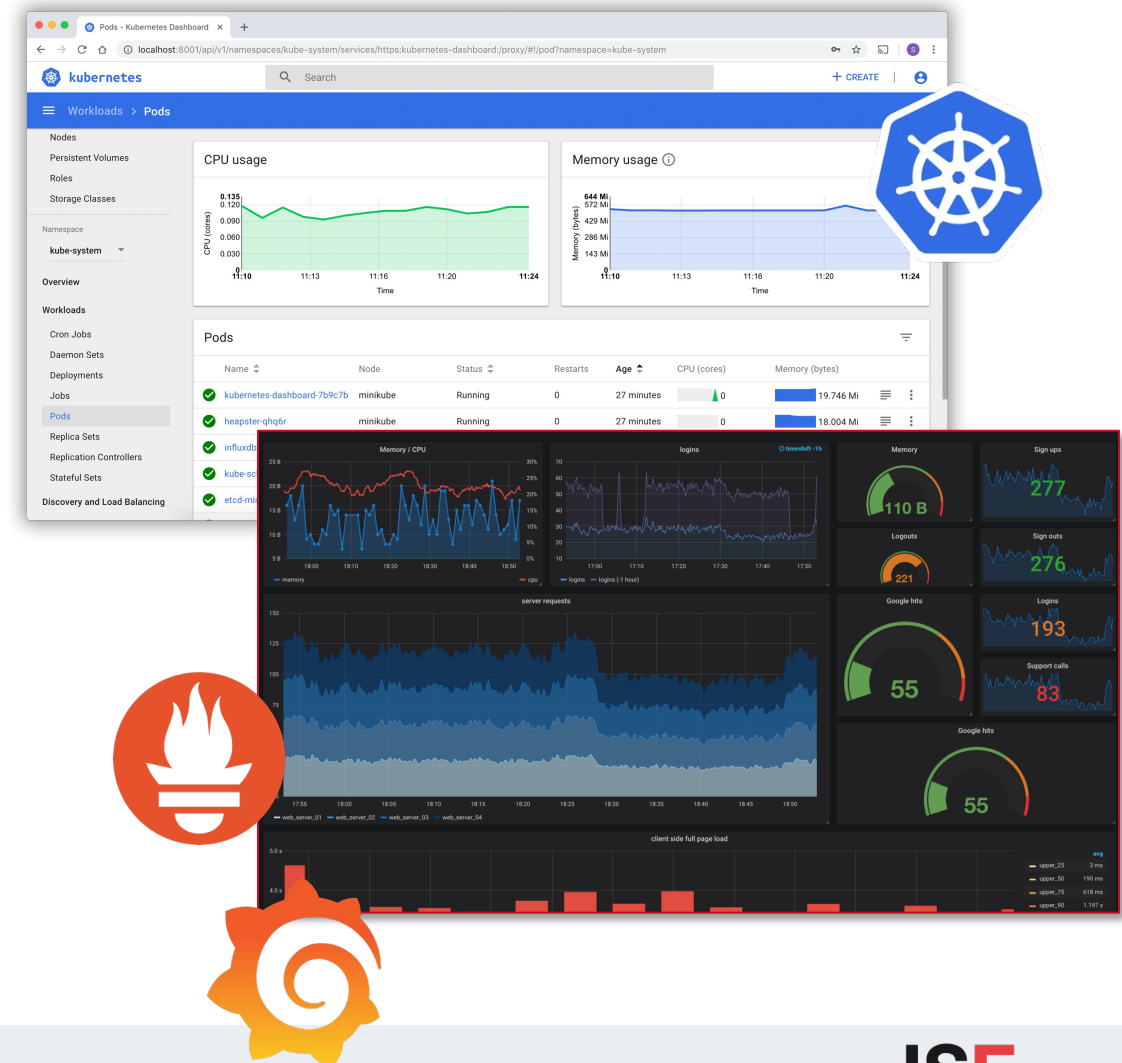
Kubernetes provides basic resource metrics “out-of-the-box”, with a metrics server add-on

Includes basic dashboard

Emits / exports metrics in “Prometheus format”, suited for Prometheus database

Custom metrics also possible with additional monitoring tools, e.g.:

- Prometheus (Time-series database + monitoring stack)
- Grafana (Visualization)
- etc..



K8s Observability - Tracing

To enable tracing in K8s, developers can leverage several tracing backends, most notably Jaeger or Zipkin

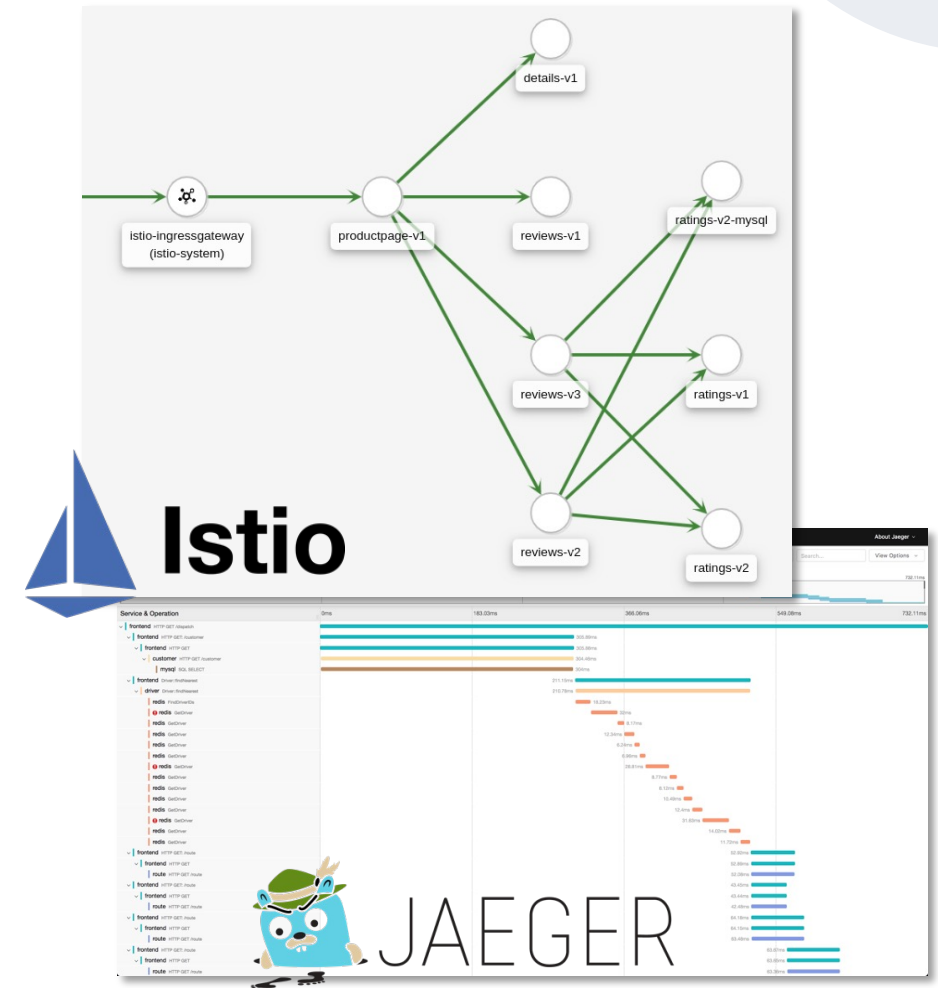
Platform-enabled tracing (e.g. with Istio):

```
kubectl label namespace default istio-injection=enabled
```

- Still requires deploying the tracing backend
- Service Topology + Dependencies can be revealed with little-to-no manual instrumentation
- Anything beyond that requires adding manual instrumentation

Developer-driven tracing (e.g. with Jaeger):

- Requires deploying the tracing backend
- Requires manual instrumentation



Serverless Observability - Monitoring & Logging

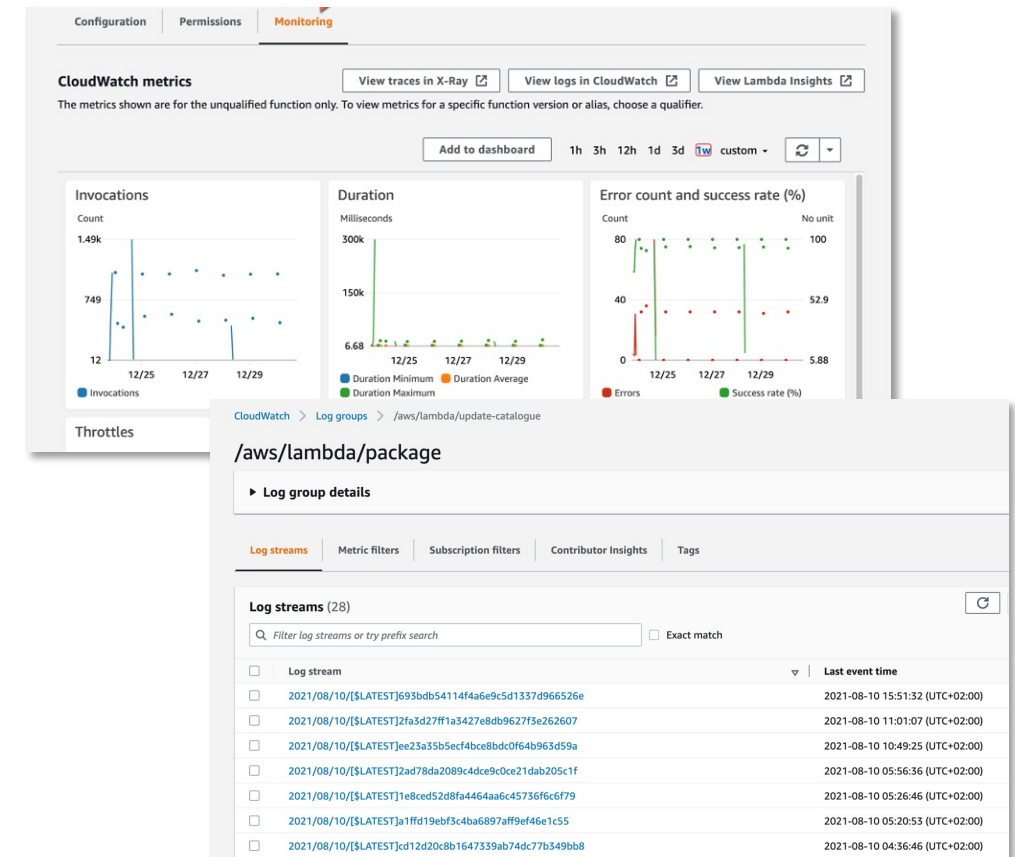
A fully-managed monitoring and logging solution is typically offered by the FaaS provider (at an additional cost..)

AWS CloudWatch Metrics:

- # of Invocations
- Concurrent executions
- Duration of functions
- Error count / success rate
- Etc..
- (Custom metrics can also be added to each function)

AWS CloudWatch Logs:

- Each function has a *log group*, which groups *log streams*
- A *log stream* is generated with each invocation of the function
- Raw logs can be exported or analyzed with yet another fully-managed tool, e.g. CloudWatch Insights



Serverless Observability - Tracing

Not every FaaS Provider supports tracing..

Two different options for FaaS, which differ in the kind of events they can trace

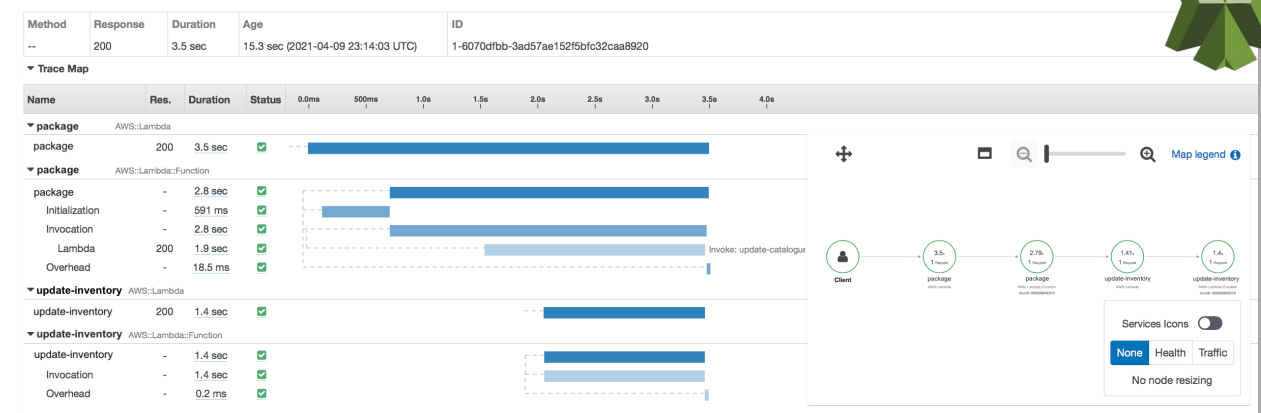
Developer-driven tracing:

- Developers can add it to the application even when the platform doesn't support tracing
- Instrumentation code is added inside each function
- Before a function can return, it must write out the tracing data to an external backend
- Example tool: Thundra.io



Platform-enabled tracing:

- The FaaS Platform offers a fully managed tracing solution
- Can trace events within functions but also has insights about the other executions steps that the platform hides from the developer, e.g. trace the start-up latency in a cold start
- Example tool: AWS X-Ray



Observing Cloud-native Applications – Challenges

Requires cross-cutting adoption within the organization

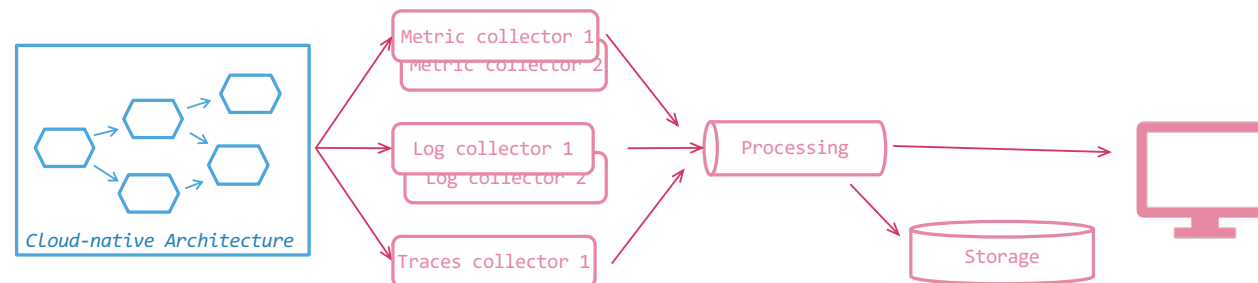
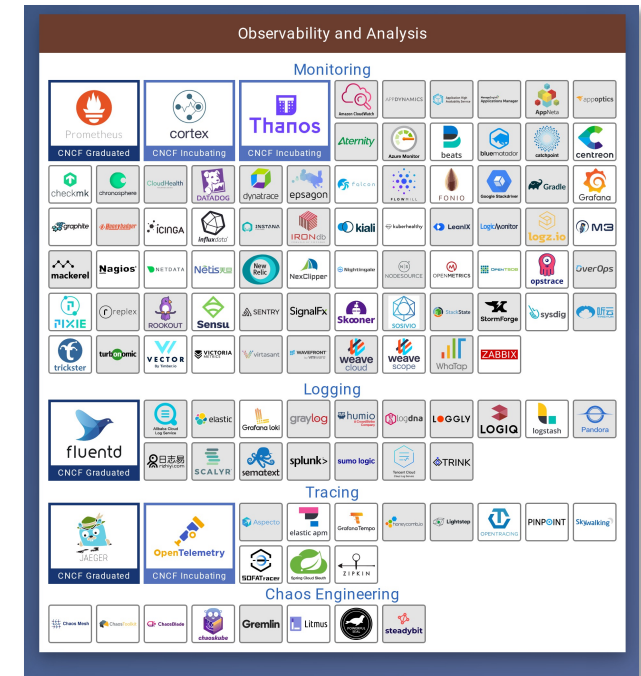
- If different teams use disparate monitoring tools that each have a specific purpose
-> tool sprawl
- Tool sprawl creates silos, costs quickly add up
- Siloed monitoring data cannot be automatically correlated to provide insights on incidents
-> slower troubleshooting

Intricate and constantly evolving stack of platforms and tools

- Configuration and Instrumentation complexity

Data management challenges

- “Pillars”/Silos make it difficult to correlate data-> Instead, move towards a unified querying, visualization and alerting approach
- Needs effective processes to generate insights through data analysis



Other Methods to Address and Ensure Runtime Quality

- Benchmarking
- Chaos Engineering
- (Testing)
- (Formal Methods)



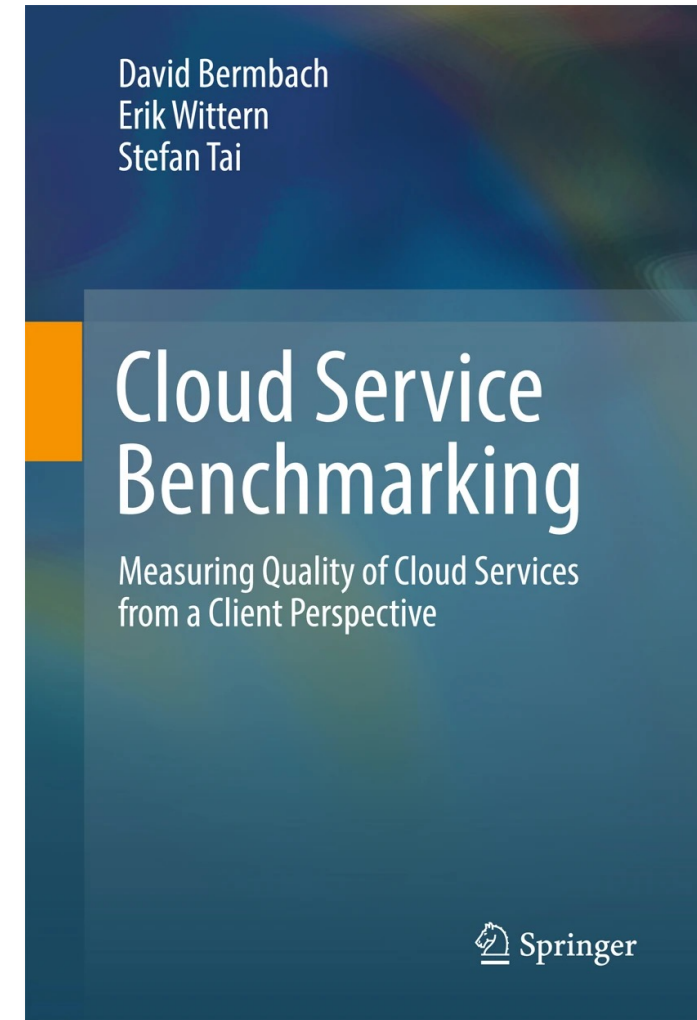
Benchmarking

Not only measure qualities, but specifically provoke **stress** situations for these qualities

Every benchmark needs a **design**: the definition of benchmark **requirements** (relevance, target audience, implementation reqs, workload reqs, etc) and **objectives** (specific (small set of) qualities and corresponding quality metrics)

Design Criteria:

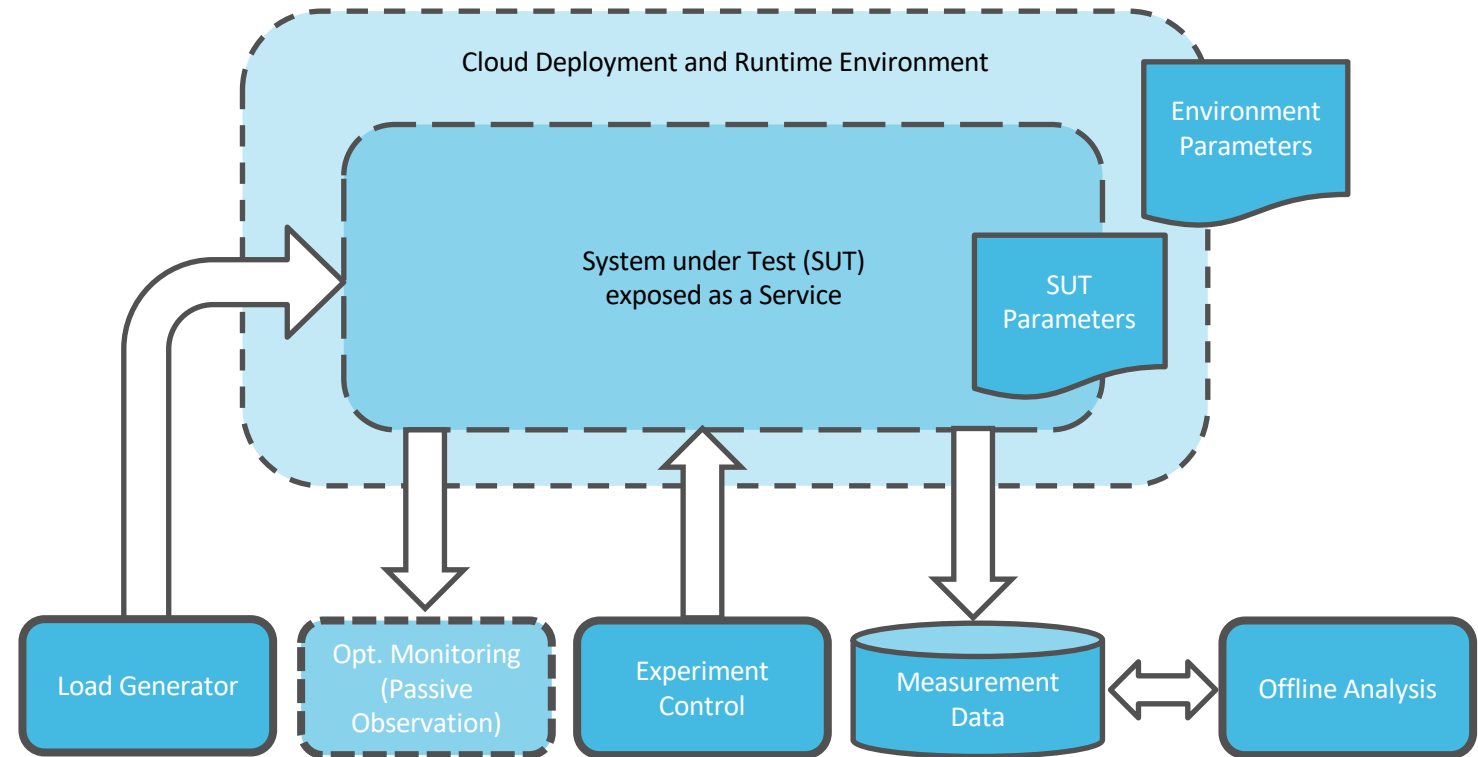
- Relevance
- Reproducibility
- Fairness
- Portability
- Understandability



Benchmarking System

Following the design phase, the benchmark *execution* deals with all matters related to *implementation* and *experimentation*. A *benchmarking system* is needed, that is, a set of components available to perform concrete measurements in support of the benchmark design. There are at least the following main components:

- A *workload generator* to generate stress on the SUT
- An *experiment controller* managing experiments
- A *measurement database* and component(s) for measuring, collecting, and storing measurement results, and to allow queries against the dataset
- An (optional) *monitoring component* to avoid bottlenecks in the measurement system



D. Bermbach, E. Wittern, S. Tai. Cloud Service Benchmarking. Springer, 2017

Benchmarking vs. Monitoring

Both **benchmarking** and **monitoring** relate to the process of measuring quality and collecting information on system states; both activities are closely related and inherently interested in the same kind of data

Category	Benchmarking	Monitoring
Motivation	Answer a specific question	Detect undesirable system states
System under test (SUT)	Test deployment	Production system
Point in time	Repeated short test runs	Permanently running process
Level of control	Complete control	Passive observation, no control
Stress on system	Artificially created	Actual production load
Impact on SUT	Strong	Negligible
Visibility delay	Often offline analysis	Near real-time

“Chaos Engineering is the discipline of experimenting on a system in order to build confidence in the system’s capability to withstand turbulent conditions in production.”

<http://principlesofchaos.org/>

Goal: harden the system by forcing it to handle failures regularly

Initial idea: randomly terminate virtual machines to make sure the system can handle failures

Build a Hypothesis around Steady State Behavior

- Focus on the measurable output of a system, rather than internal attributes of the system
- Measurements of that output over a short period of time constitute a proxy for the system's steady state
- The overall system's throughput, error rates, latency percentiles, etc. could all be metrics of interest representing steady state behavior
- By focusing on systemic behavior patterns during experiments, Chaos verifies that the system does work, not how it works

Vary Real-world Events

- Chaos variables should reflect real-world events
- Prioritize events either by potential impact or estimated frequency
- Consider different events: hardware failures, software failures like malformed responses, and non-failure events like a spike in traffic or a scaling event
- Any event capable of disrupting steady state is a potential variable in a Chaos experiment

Chaos Engineering Principles II

Run Experiments in Production

- Systems behave differently depending on environment and traffic: utilization can change at any time, so sampling real traffic is the only way to reliably capture the request path
- To guarantee both authenticity of the way in which the system is exercised and relevance to the current deployed system, Chaos strongly prefers to experiment directly on production traffic

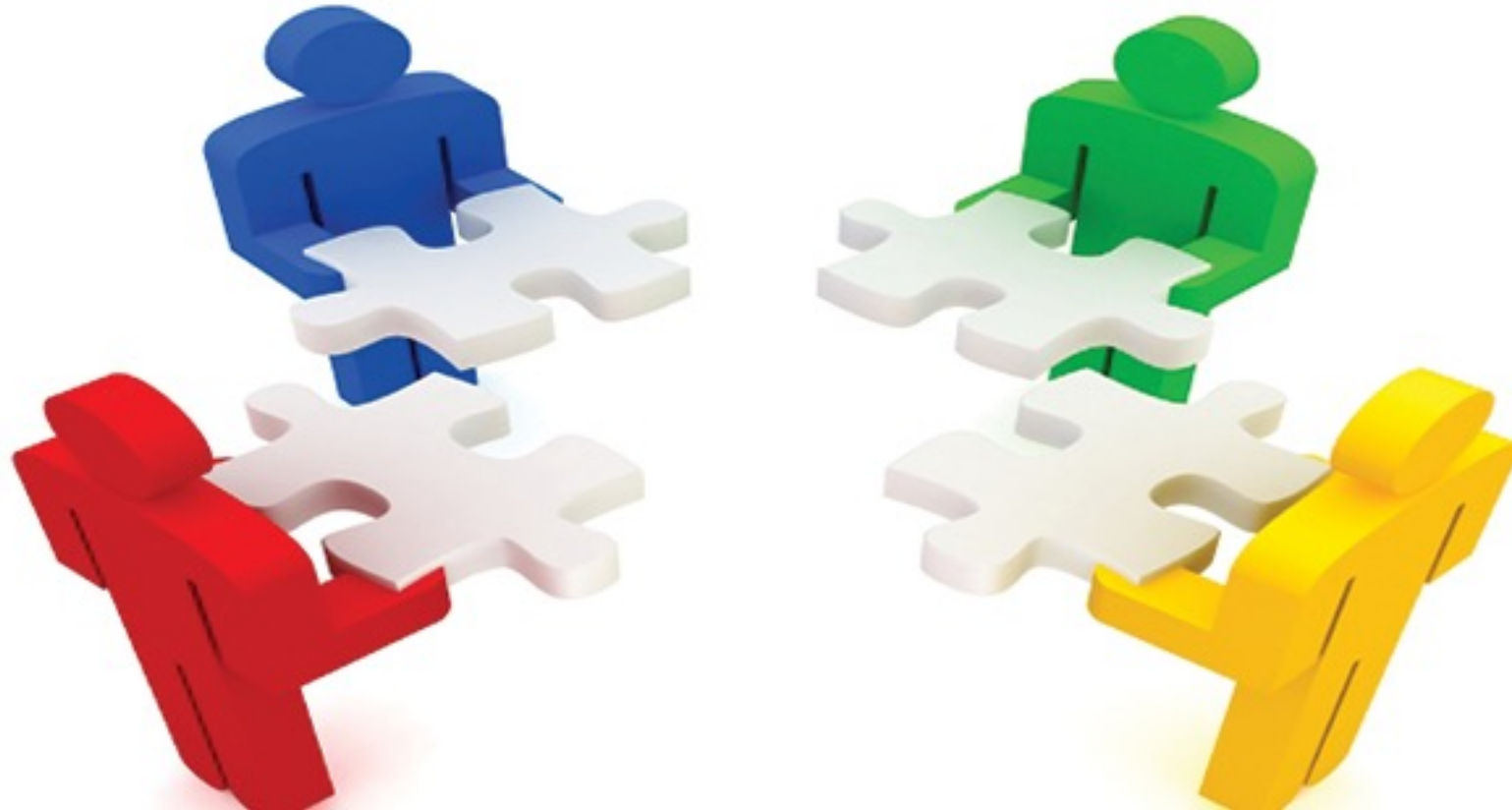
Automate Experiments to Run Continuously

- Automate experiments and run them continuously to reduce manual labor and sustain them indefinitely
- Chaos Engineering builds automation into the system to drive both orchestration and analysis

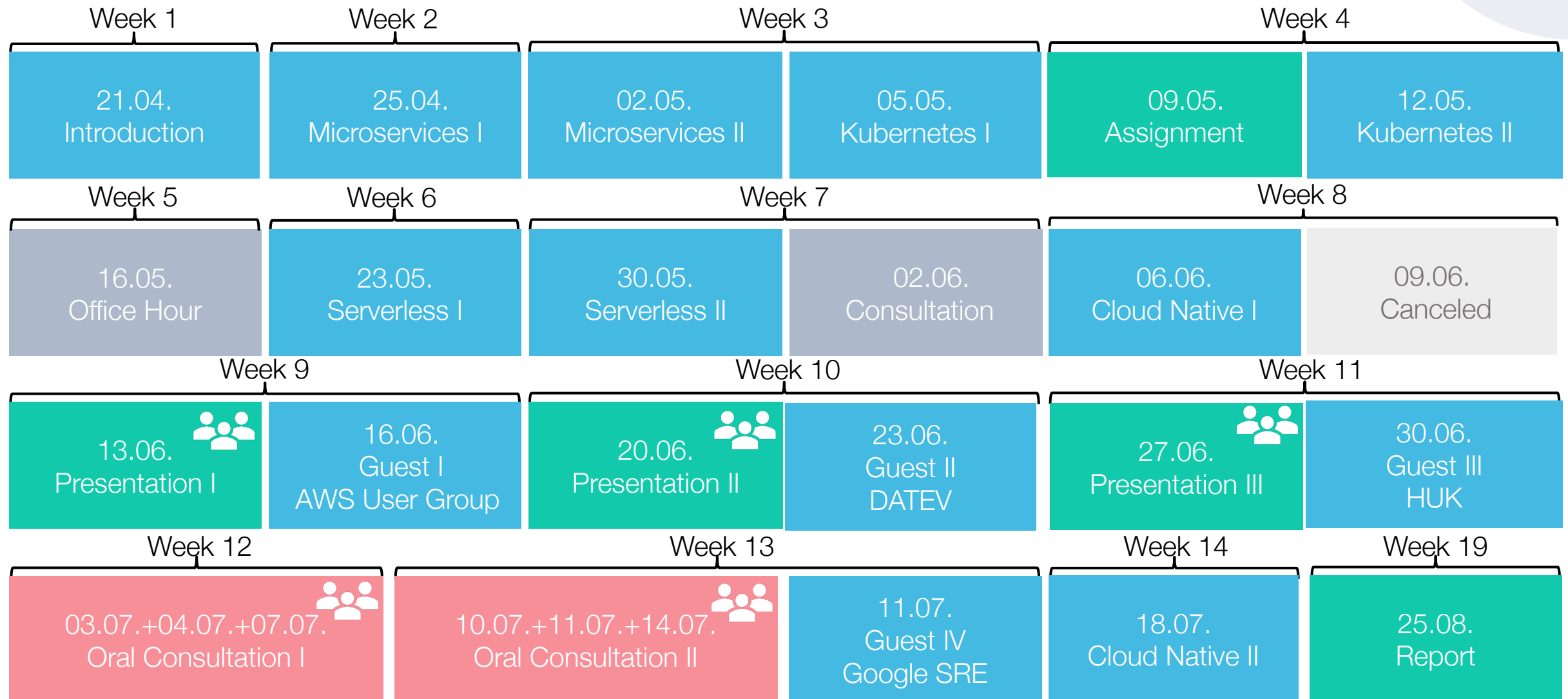
Minimize Blast Radius

- Experimenting in production has the potential to cause unnecessary customer pain.
- Allow short-term negative impact, but ensure that the fallout from experiments is minimized and contained

Being Cloud-Native, not Naive... Bringing it all Together



Schedule



Upcoming Deadlines

Design Document and **Presentation Slides** are due next week

Presentation:

- 10 min presentation
- 10 min discussion

Deadline: 12.06.2023 at 10:00am

Design Document:

- 4-6 Pages incl. references
- Format: IEEE Double Column
- 25%-50% generative elements
- Appendix: “creation log” – prompts you used, output you got and where you used the output in the text
- Brief description of who contributed what

Deadline: 12.06.2023 at 23:59

→ Upload via ISIS

